Considering Browser Interaction in Web Application Testing[1]

Giuseppe A. Di Lucca, Massimiliano Di Penta
(dilucca, dipenta)@unisannio.it

* RCOST - Research Centre on Software Technology
Department of Engineering, University of Sannio
Palazzo ex Poste, Via Traiano - 82100 Benevento, Italy

Abstract

As web applications evolves, their structure may become more and more complex. Thus, systematic approaches/methods for web application testing are needed. Existing methods take into consideration only those actions/events the user is prompted by the application itself, such as the selection of a hypertextual link or the submission of the data contained in a form. However, these methods do not consider also actions/events provided by the browser, such as the usage of backward and forward buttons, usage that in some cases may produce navigation inconsistencies.

This paper proposes an approch to integrate existing testing techniques with a state-based testing devoted to discover possible inconsistencies caused by interactions with web browser buttons. A testing model, considering the role of the browser while navigating a web application, and some coverage criteria, are presented.

Keywords: Web application testing, State-based testing, Software testing.

1. Introduction

The complexity of web applications (WAs) has considerably grown in recent years: initially conceived for simple tasks, such as handling page access counters or guestbooks, WAs have now became a way to develop complex and critical software systems. E-commerce applications, but also scientific or medical applications, are just some examples. A consequence is that WAs are expected to meet higher quality requirements, in particular a high reliability is requested.

Several different approaches have been proposed for WA testing. Existing commercial tools focus on aspects such as performance or broken link detection. Recently, few approaches have been proposed for structural and functional testing. Ricca and Tonella [1] proposed essentially a white box approach based on a UML model of WAs. Data flow testing of WAs was discussed in [2]. Di Lucca et al. [3] proposed a black and a white box testing approach based on a testing model for WAs, derived from Conallen's model [4], and a tool, named WAT, to support the different phases of the testing.

All in all, a WA is a distributed system: most of the existing testing techniques can be customized for testing WAs. However, WAs have some peculiarities that make them different from traditional (even distributed) software systems. Most of these challenges were discussed in [5], [6]: testing a WA is challenging, not only for the mission critical tasks it carries out, but also for factors such as the security, the scalability, and the high dependency of the outputs on the browser in which pages are rendered and on the way a user interacts with the browser itself.

The interaction with the browser, and the way the WA state is preserved are two factors worth of a deeper investigation. In fact:

- The user interface of a WA is the page rendered in a web browser. The user can interact with the WA by means of both of controls/actions/events provided by the WA itself and by any browser control. Most of the existing testing techniques focused on the interaction with HTML forms, without considering what happens when a user interacts with browser buttons (such as backward, forward, reload); and
- A WA is, from a simpler point of view, stateless. However, there are some mechanisms, such as cookies and sessions variables, to keep its state. This aspect has been considered in [3], then extensively covered by Elbaum et al. in [7].

[1] Draft to appear in WSE 2003

Given this, we propose to integrate structural or functional WA testing approaches, or even approaches exploiting session data, with a state-based testing of WA in the browser context.

We propose to model the behavior of a web browser by a statechart to investigate on how the browser states may affect a user navigation. Given such a statechart, and given a navigation path of the WA, identified by a test case derived from any testing strategy, we will show, by some examples, how the browser states may affect the navigation and in which way they have to be considered when testing a WA. The proposed approach mainly focuses on discovering failures caused by some sequences of interactions with browser buttons, independently from the type of browser used.

The paper is organized as follows. Section 2 discusses the motivations of the approach, identifying some possible inconsistencies that can be found by exercising the WA in the context of a browser. Section 3 describes the testing model and the statechart of a web browser. Section 4 presents the proposed testing approach, and discusses how it can be integrated with existing strategies, which coverage criteria can be adopted and how to define constraints for specifying oracles. The last section summarizes conclusions and identifies directions for future developments.

2. Motivations

Often a user, while navigating through a WA, besides to follow the WA links to reach a target page, uses the browser buttons, i.e., the back(ward), forward or reload buttons, to re-display some of the pages already visited along the navigation. The usage of those browser elements may negatively affect the navigation, because they might introduce some inconsistencies (or violate any functional/not-functional WA requirements). Some common examples are:

- The user fills up a form, clicks the submit button and reaches the next page; if he/she goes back using the *back(ward)* button, then the inputted data might not be displayed again in the form;
- Similarly, let us suppose we have a form to insert/edit data for a new product, a new customer, or whatever. When the user wants to modify data, then the form fields are filled up with the data stored in the database. This can be done in several ways, for example:
    1. By dynamically generating the "`value`" /"`selected`" attributes of input and select fields HTML tags (and others);
    2. Using an automatically generated Javascript function that catches the event `onInit()` and then fills up the form.

  Inconsistencies may happen, due to navigation, in the second case. If the user modifies the data in the form (without submitting them), goes back to the previous page and then returns again to the initial page, then the Javascript function will reload the old data, discarding all the changes made. Attention should be therefore paid to verify if this meets the WA requirements;
- When using a web-mail system, the user selects a message in the incoming list, then the text of that message is displayed; if the user returns to the incoming list page using the back button, the previously displayed message is not checked as "*read*";
- If a user reaches a page containing an access counter, and then he/she uses just the back and forward buttons to re-display the page, then the value of the counter might not be incremented[2];
- When a user authenticates himself/herself by submitting a username and a password, if during the navigation he/she returns to the login page and then goes forward simply using the browser buttons, then the session should expire.

In all of the above cases we can note some possible inconsistencies, or that WA requirements could not be met.

Of course, all possible inconsistencies are to be found with respect to the WA specifications. For the first example above mentioned (filling a form), such a behavior can be just the one specified for that particular functionality, i.e., the specifications said that forms must be cleared in case of backward navigation performed using the browser button.

To avoid that the usage of browser features may negatively affect the navigation of a WA, such features should be worth of consideration when testing the behavior of the WA. In particular, each specified functional behavior should be tested by considering it together with each possible browser feature.

The testing model of the WA should therefore model the browser, considering all its features that are relevant from a testing point of view.

3. The Testing Model of a Web Browser

Existing WA testing approaches, such as those proposed in [1], [2], [3], and [7] aim to generate test cases for covering *link navigation paths* in a WA, where a *link*

[2] This example does not consider page counters counting once more requests coming from the same IP address in the same time interval.

navigation path is defined as a path from a starting page to a final page just following a given set of hyperlinks provided by the WA itself.

Each time a form is encountered, the input fields are filled with test data and a submit action is performed. For sake of generality, we will refer to both hyperlinks and submit actions as *links*. When links are followed, the browser builds a sequence of visited pages. Let `n` be the length of this sequence, we indicate with `pageno` the page position in the sequence, i.e., `pageno=1` for the first page, `pageno=n` for the last page.

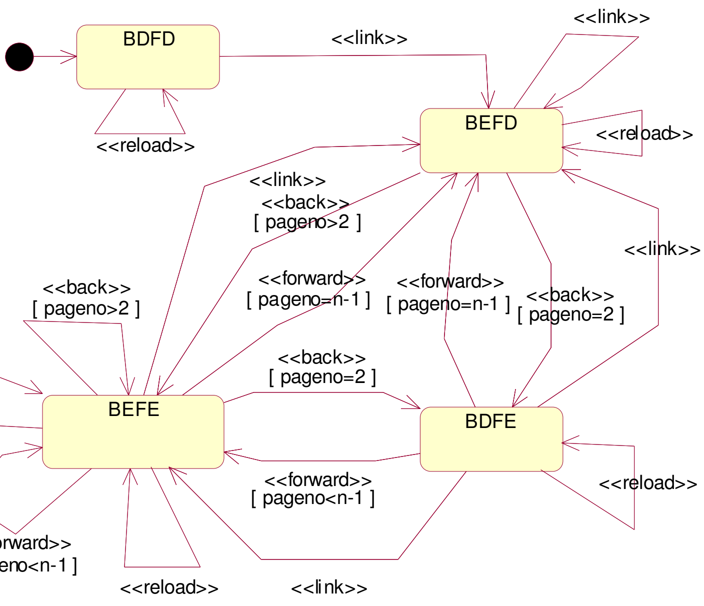

Figure 2 – Statechart of a web browser

As stated in the previous section, covering such a *link navigation path* does not take into account the fact that a user may perform navigation also using the browser buttons (*Reload*, *Forward* and *Back*).

Given this, a browser is characterized by the web page displayed, by the state of its buttons (*Enabled* or *Disabled*), and by the sequence of visualized pages, also navigable using the browser buttons. We can model all these features by a statechart, where each state is defined by the page displayed and by the state of the buttons, while the user actions on page links or browser buttons determine the state transitions.

With respect to the buttons, we can identify the following states:

1. *Back Disabled, Forward Disabled (BDFD)*: this is the state for the first page visualized by the browser; the state invariant [9] is `pageno=1 and n=1`;
2. *Back Enabled, Forward Disabled (BEFD):* this happens when the browser visualizes the last (but not the first) page visited; the state invariant is `pageno=n and n>1`;
3. *Back Enabled, Forward Enabled (BEFE):* this happens when the page visualized is not the first, nor the last in the navigation sequence; the state invariant is `pageno>1 and pageno<n`;
4. *Back Disabled, Forward Enabled (BDFE):* this happens when the page visualized is the first but not the last in the visualization sequence; the state invariant is `pageno=1 and n>1`.

Transitions between the four states are activated by the pressure of the three buttons *Reload*, *Back*, *Forward*, or by following a *link*. Figure 1 reports the statechart of a web browser; each state is named by the state of the *Back* and *Forward* buttons (i.e., the B and F characters to indicate *Back* and *Forward* button respectively, and the D end E characters to indicate the *Disabled* and *Enabled* button states respectively). The state transitions are labeled with the actions (corresponding to events in the statechart model) a user can make to navigate between web pages, and with guard conditions. For example, `<<back>> [pageno=2]` means that the event *back* can happen iff the page displayed in the browser is the second one in the navigation sequence (in that case after a back is performed, the back button not enabled anymore since the reached page is the first one in the navigation sequence).

To improve the figure readability, the statechart does not show a final state: it is worth noting that such a state can be reached from any other state when the user closes the browser.

A path from an initial page to a final page, that involves interactions with browser buttons, is simply called a *navigation*. As it will be shown in the next sections, testing a WA in a browser context implies exercising such *navigations*.

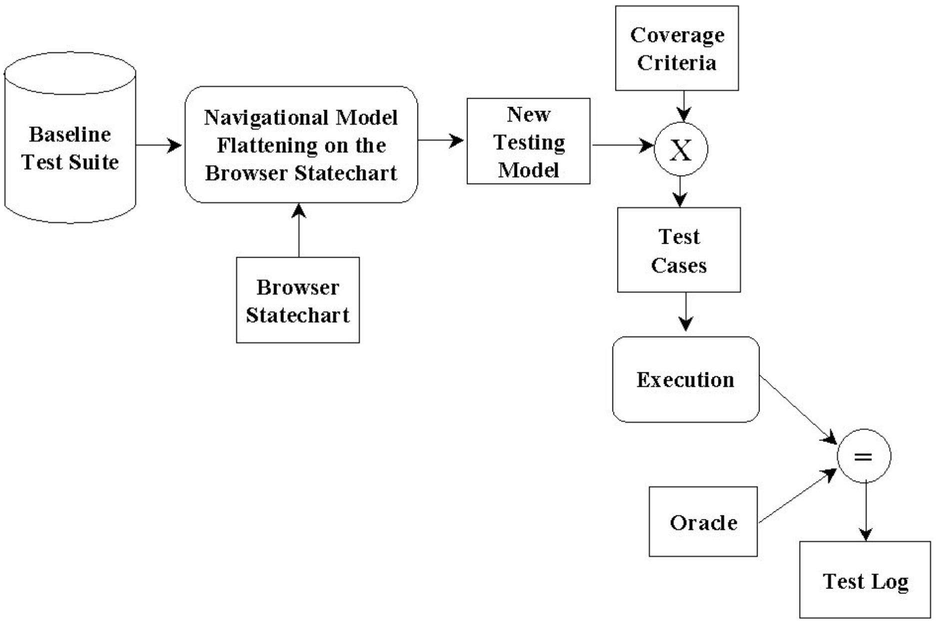

Figure 3 – The proposed approach

An interesting situation is when a link inside a page (say Page A) causes the opening of a new browser window (i.e., by means of a "`_new`" targeted link), displaying another page (say Page B). In this case we will have the instantiation of a new statechart (obviously in the BDFD state) modeling the interaction with the new browser window. A composite statechart can be used to take into account the interactions between the old and the new browser windows. Such a composite statechart can be flattened by means of a "`_new`" link connecting the current state of Page A with the BDFD state of Page B.

Finally, it is worth noting that the browser model can be further on enriched also considering other interesting aspects that also may affect a user navigation, such as:

- Enabling/disabling client-side scripts (e.g., Javascript);
- Enabling/disabling cookies; or
- Browser visualization capabilities, etc.

Although the idea and the model presented can be easily adapted to consider such features, an extensive coverage of them is out of the scope of this paper.

4. The proposed testing approach

The proposed approach is mainly focused on the way browser interactions may affect a user navigation, thus it may exploit existing testing approaches to test functional/structural features of a WA. At the same time, it defines a way to integrate such an approach with the statechart describing the browser test model defined in the previous section. The proposed approach consists of the following main steps:

1. Select/define a testing approach among the existing ones such as those proposed in [1], [2], [3] and [7];
2. Define a set of test cases according to the selected approach. These test cases, named *baseline test cases*, compose the *baseline test suite* for the WA;
3. Flatten each *baseline test case exercising a link navigation path* with respect to the browser statechart described in Section 3;
4. Define testing coverage criteria on the flattened statechart, and consequently produce a test suite satisfying such criteria. The test case produced are named *browser test cases*; and
5. Execute the *browser test cases* and compare the results of the execution with an oracle [8] for detecting potential inconsistencies due to a given sequence of interactions. The coverage of the chosen criteria must also be verified.

4.1 Flattening the Statechart

To perform the browser state-based testing, the browser statechart must be flattened with respect to the navigation path followed by each *baseline test case*. Let us suppose, for example, that a *baseline test case* exercises the link navigation sequence composed of the pages A, B, C. Just considering the pages displayed in the browser window, the browser crosses three states: *Page A*, *Page B*, *Page C* (see Figure 5).

The statechart flattening can be performed similarly to what is done for the statecharts of class hierarchies when using state-based testing on subclasses (see [9] for further details).

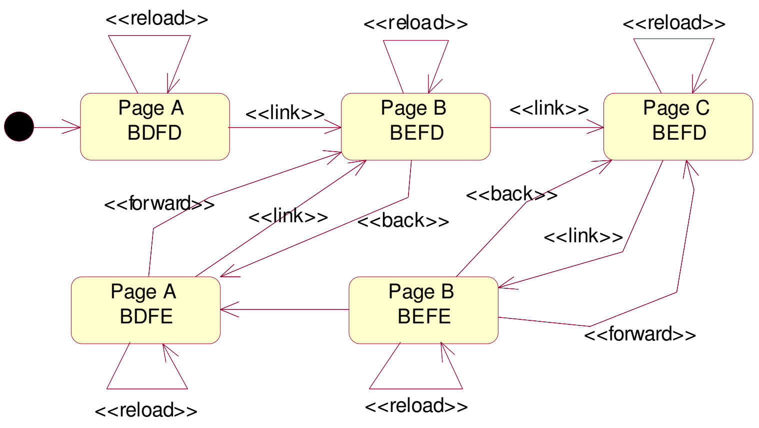

Figure 4 – Flattened statechart

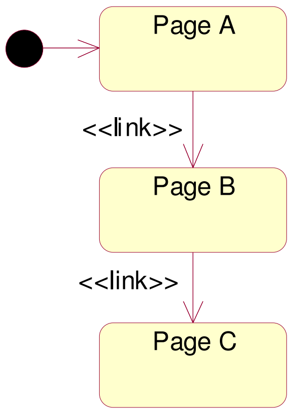

Figure 5 – Statechart of the browser with respect to the pages visualized

When flattening the statechart, it is worth noting that:

1. Only for the first page in the sequence both the forward and back buttons can be disabled at the same time;
2. The first page can never have the back button enabled; and
3. The last page can never have the forward button enabled.

The flattened statechart of this example is shown in Figure 4.

For a sequence of length n, the flattened statechart will have 2 (n-1) + 1 states. The latter expression highlights the linear relationship between the number of pages composing the WA and the number of states in the flattened statechart. Thus scalability is not negatively affected. Even if the *base test case* contains more accesses to the same page (e.g., the sequence ABA), the above considerations are still valid.

4.2 Integration with Existing Approaches

This section presents an example describing how existing testing approaches can be integrated with the browser state model. The testing approach proposed in [3] is exploited to define the *baseline test cases*.

The approach proposed in [3] defines a *test strategy* to generate test cases from a specified test model, as well as various *testing levels* for a WA, specifying the different scopes of the tests to be run. In particular the strategy is based on the category partition testing technique [10], where the input data are divided into a set of equivalence classes and all the variant combinations of them are considered, while the different kinds of units to test and their integration is driven by the specified test model.

The integration of such a technique with the one based on the browser state will require that each variant, considered to test a unit or a group of units, has to be tested for each browser state and state transition.

Let us suppose we have to test the function provided by a WA allowing a user to login. In this case a *LoginPage* will be displayed, requiring *UserId* and *Password.* After this data has been submitted, a *LoggedPage* will be displayed if both *UserId* and Password are correct. On the contrary, a *LoginErrorPage* containing an error message is displayed if at least one of the submitted values for *UserId* and *Password* is wrong. Moreover, each time the *LoginPage* is displayed, both the *UserId* and *Password* fields must be empty. Finally, from the *LoginPage*, it is possible to go to the next page by just pressing the submit button.
According to [3], and focusing the attention just to the client side of the WA, the set of test cases to be associated with the *LoginPage* are described by a table such as Table 1, where the *baseline test cases* for the described functions are reported together with the expected results.

Table 1: Baseline test cases for the example

Variant	Input variables		Expected Results
	UserID	Password	
1	Omitted	DC	LoginErrorPage
2	DC	Omitted	LoginErrorPage
3	Correct	Correct	LoggedPage
4	Incorrect	DC	LoginErrorPage
5	DC	Incorrect	LoginErrorPage

Note: DC = Don't Care

If we consider just the WA links, only the navigation from the *LoginPage* to the *LoggedPage* or the *LoginErrorPage* could be possible. By considering also the browser interactions we could have the following navigations starting/ending from/to the *LoginPage*:

- The *LoginPage* is the first page displayed in the browser (`pageno=1, n=1 -> BDFD`); the user correctly fills the *UserId* and *Password* fields (*Variant #3*) and submits them; the *LoggedPage* is displayed (`pageno=2, n=2 -> BEFD`); the user presses the *Back* button; the LoginPage is displayed again (`pageno=1, n=3, -> BDFE`);
- The *LoginPage* is the first page displayed in the browser (`pageno=1, n=1 -> BDFD`); the user incorrectly fills the *UserId* or the *Password* fields (*Variant* #1, *#2*, *#4*, or *#5*) and submits them; the *LoginErrorPage* is displayed (`pageno=2, n=2 -> BEFD`); the user presses the *Back* button; the *LoginPage* is displayed again (`pageno=1, n=3 -> BDFE`);
- The *LoginPage* is not the first page displayed in the browser (`pageno=n, n>1 -> BEFD`); the user fills the *UserId* and *Password* fields (any Variant); the user presses the *Back* button returning back to the last page displayed (`1≤pageno<n-1, n>1 -> B?FE`[3]); the user presses the *Forward* button; the *LoginPage* is displayed again (`pageno=n, n>1 -> BEFD`);
- The *LoginPage* is not the first page nor the last one displayed in the browser (`1<pageno<n, n>1 -> BEFE`); the user fills the *UserId* and *Password* fields (any Variant); the user presses the *Forward* button; the *LoginPage* is displayed again.

Table 2 summarizes the above described navigation indicating the pages and state transitions involved.

Table 2: State transitions for the example

Precondition and Starting State	Current Page	User Action	Next Page and Resulting State
pageno=1, n=1 BDFD	LoginPage	Variant=3; Link	LoggedPage BEFD
pageno=2, n=2 BEFD	LoggedPage	Back	LoginPage BDFE
pageno=n, n>1 BEFD	LoginPage	Variant =DC; Back	Previous Page BEFE
pageno=n-1 BEFE	Previous Page	Forward	LoginPage BEFD
DC	LoginPage	Reload	LoginPage No state transition
pageno=1, n=1 BDFD	LoginPage	Variant= 1,2,4,5; Link	LoginErrorPage BEFD
pageno=2, n=2 BEFE	LoginErrorPage	Back	LoginPage BDFE
pageno=1, n>1 BDFE	LoginPage	Variant = 3; Forward	LoggedPage BEFE
pageno>1, n>1 BEFE	LoginPage	Variant = 1,2,4,5; Forward	LoginErrorPage BEFE

4.3 Defining Coverage Criteria

Once defined a testing model, the generation of test cases, as well as the effectiveness assessment of a given test suite, can be performed according to a properly defined coverage criterion. Since a flattened statechart is generated from each *baseline test case*, any coverage criterion must be inclusive of the criterion used to generate the *baseline test suite.*

[3] B?FE means that if the target page is the first one visualized, then the resulting state is BDFE, otherwise it is BEFE.

The test case execution may be fully automated, by developing a tool, whose inputs are the *baseline test cases* and the chosen coverage criteria, that simulates the browser navigation.

Testing coverage criteria can be derived from some of those proposed by Binder [9] for object-oriented systems (we report them in increasing order of strength):

1. All states;
2. All transitions;
3. All transition k-tuples;
4. All round-trip paths.

The meaning of the first two strategies is quite simple; the third implies the coverage of all possible sequences of transitions having length *k* (it is similar to the k-coverage criteria of cycles in a control flow graph). Finally, a round-trip coverage [9], [11] is obtained when each sequence of transitions beginning and ending in the same state is exercised at least once. For the statechart in Figure 4 the roundtrip strategy implies covering the transition tree [9] shown in Figure 6.

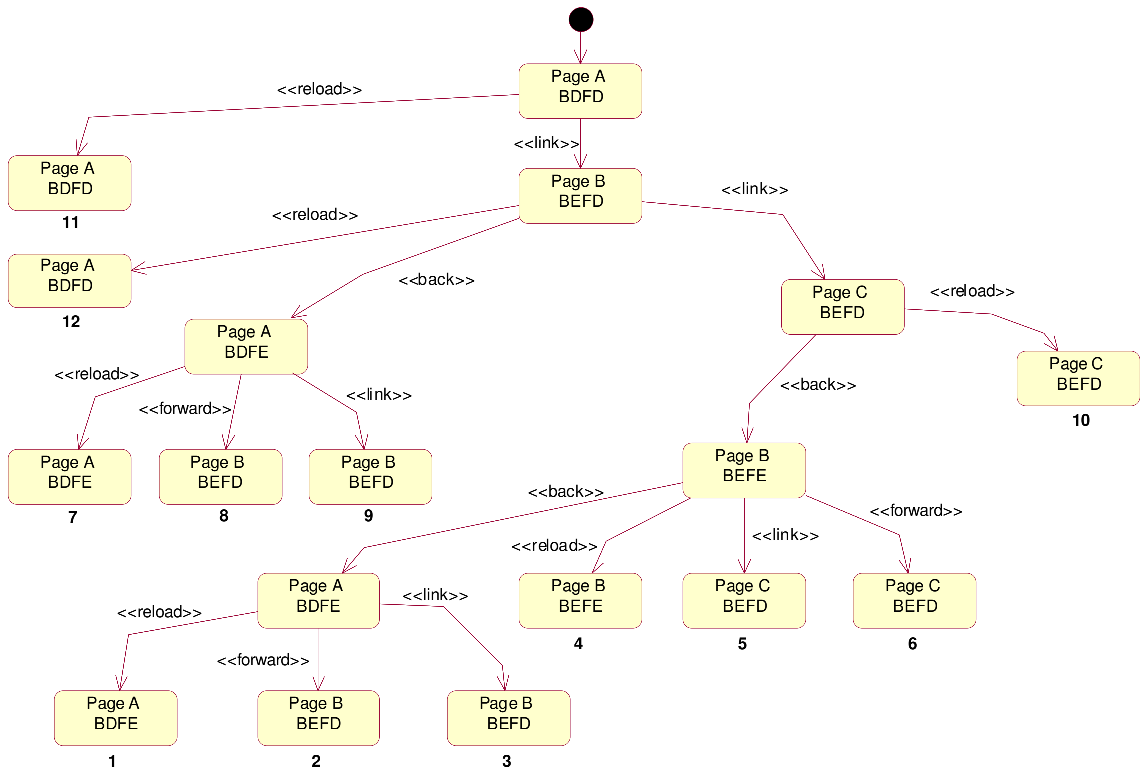

Figure 6 - The transition tree for the statechart in Figure 4

For traditional object-oriented systems, literature suggests to adopt, at least, the all-transitions strategy: any cheaper strategy could not reveal a significant number of failures. These considerations are still valid for our approach, since, for example, all the states in the statechart of Figure 4 can be covered using a single test case that follows the *navigation*[4]:

```
Page A (BDFD) – <<link>> – Page B
(BEFD) – <<link>> – Page C (BEFD) –
<<back>> – Page B (BEFE) – <<back>> –
Page A (BDFE)
```

[4] Labels enclosed in guillemots indicate the events activating transitions (e.g., `<<link>>`), while other labels indicate the states.

In this case, some relevant failures may be discovered, in that the longest possible sequence of back actions has been followed: inconsistency in the content of forms, or the expiration of a session are just some examples. It is worth noting that the latter could not be an undesired behavior: in WAs where password is required for authentication, the sessions are supposed to expire, due to security reasons, when following such a sequence.

The round-trip strategy, even if more expensive than others, is also the one that can discover the largest number of failures. Whenever possible, appropriate heuristics can be used to reduce the number of test cases. In most situations, a *reload* should not, for example, produce an undesired effect; for the transition tree of Figure 6, six paths can be pruned out. Different heuristics can be adopted according to other WA specifications.

4.4 Specifying the Oracle

The proposed WA testing approach offers the possibility of defining rules for the automatic detection of some inconsistencies.

The oracle may be defined producing some assertions on the navigation sequences. For examples, if the user follows the sequence:

```
Page A – <link> – Page B – <link> –
Page A
```

or the sequence:

```
Page A – <<link>> – Page B – <<back>>
– Page A
```

then, given the application requirements, it is possible to assert that, in the first case, the visualization of `Page A`, at the final step, may differ from its visualization at the first step. In the second case, we will obtain the same visualization for `Page A`. This behavior resembles what discussed in Section 2 about the web mail system. Another example is when `Page A` contains an access page counter. In that case it is also possible to assert, for example, that for the sequence:

```
Page A – <<reload>> – Page A
```

the two visualizations of `Page A` differ.

5. Conclusions and work-in-progress

This paper proposed an approach to complement WA testing strategies, such as black box, white box or even strategies exploiting session data, with a state-based testing to verify the correct interaction of the WA with a web browser. The aim is to discover failures due to particular sequences of navigations performed using the web browser buttons.

The test case generation can be automated once *baseline test cases*, produced with other "traditional" approaches, are available. Also the test execution can be automatically performed developing a tool that simulates the actions of a web browser, with the ability of going forward, backward and reloading a page. Finally, heuristics can be adopted to automatically reveal some inconsistencies caused by following particular sequences of actions to navigate among pages.

Some aspects still remain to be investigated. For example, an extended model should consider if the cookies are enabled or not, if client side scripting is enabled or not, visualization capabilities (e.g., frames), etc. Last, but not least, a real challenge: the response of the WA to the interactions with the browser is *time dependent*. In other words, if a session expires while the user is filling up a form, he/she will not be able to complete the transaction.

Work-in-progress is devoted to develop a tool to generate test cases for the state-based coverage criteria, as well as to automate the execution of test suites. Experiments will also be performed to assess the effectiveness of the approach.

Acknowledgments

This work has been partially supported by the project"Software Architectures for Heterogeneous Access Networks Infrastructures – SAHARA" funded by "Ministero dell'Istruzione, dell'Università e della Ricerca Scientifica- MIUR".

References

[1]. F. Ricca and P. Tonella, "Analysis and Testing of Web Applications". *In Proceedings of 23th IEEE International Conference on Software Engineering*, Toronto, ON, Canada, May 2001.

[2]. C. Liu, D. Kung, P. Hsia, and C. Hsu. "Structural testing of web applications". *In Proceedings of the 11th IEEE International Symposium on Software Reliability Engineering*, pages 84-96, Oct. 2000.

[3]. G. A. Di Lucca, A. R. Fasolino, F. Faralli, U. De Carlini, "Testing Web Applications". *In Proceedings of the IEEE International Conference on Software Maintenance*, pages 310-319, Oct 2002, Montréal, QC, Canada.

[4]. J. Conallen, *Building Web Applications with UML*, Addison Wesley, 2000.

[5]. G. A. Stout, "Testing a Website: Best Practices", whitepaper on www.reveregroup.com.

[6]. E. Hieatt, R. Mee, "Going Faster: Testing The Web Application". *IEEE Software*, Mar. 2002, pp. 60- 65.

[7]. S. Elbaum, S. Karre, G. Rothermel, “Improving Web Application Testing with User Session Data”. *In Proceedings of the 25th IEEE International Conference on Software Engineering*, pages 49-59, May 2003, Portland, OR, USA.

[8]. W. Howden and P. Eichhorst. Proving properties of programs for program traces. In *Tutorial: Software Testing and Validation Techniques*; E. Miller and W. E. Howden ed., New York, IEEE CS Press, 1978.

[9]. R. Binder. *Testing Object-Oriented Systems*. Addison Wesley, Reading, MA, 2000.

[10]. T.Ostrand, M. Balcer, “The category-partition method for specifying and generating functional test”. *Communications of the ACM*, 31(6), June 1988.

[11]. G. Antoniol, L. C. Briand, M. Di Penta, Y. Labiche “A Case Study Using the Round-Trip Strategy for State-Based Class Testing”. *In Proceedings of the 13 International Symposium on Software Reliability Engineering* – November 2002 – Annapolis – Maryland (USA).